Recording and Refactoring HOL Light Tactic Proofs

Recording and Refactoring HOL Light Tactic Proofs

Mark Adams, David Aspinall
School of Informatics, University of Edinburgh

June 10, 2012

Abstract. In this article we present a mechanism for recording HOL Light tactic proofs in a hierarchical tree structure, with information stored at the level of atoms in the user's ML proof script. This is written to support refactoring of tactic proofs, so that single-tactic packaged-up proofs can be flattened to a series of interactive tactic steps, and vice versa. It also provides a good basis for proof visualisation and for proof querying capability. The techniques presented can be adapted straightforwardly to other systems.

1 Introduction

Although now 30 years old, Paulson's subgoal package [1] is still the predominant mode of interactive proof in various contemporary theorem provers, including HOL4, HOL Light and ProofPower. The Flyspeck Project [2], a massive international collaborative effort to formalise Hale's Kepler Conjecture proof, uses HOL Light's subgoal package throughout, for example.

The subgoal package is simple in concept and yet remarkably effective in practice. Users start with a single main goal, which gets broken down over a series of tactic steps into hopefully simpler-to-prove subgoals. The user focuses on each subgoal in turn, moving onto the next when the current has been proved. The proof is complete when the last subgoal has been proved. Behind the scenes, the subgoal package is keeping everything organised by maintaining a proof state that consists of a list of current proof goals and a justification function for constructing the formal proof of a goal from the formal proofs of its subgoals. Tactics are functions that take a goal and return a subgoal list plus a justification function. The subgoal package state is updated every time a tactic is applied, incorporating the tactic's resulting subgoals and justification function.

Despite its widepread use, subgoal package user facilities remain basic and lack useful extended features. One potentially useful facility is automated proof refactoring, which can help save time for both experts and novices. Others include proof graph visualisation, which can help the user understand a large proof, and proof querying, for answering basic questions about the nature of a proof, such as the tactics and theorems it uses.

In this article we first motivate the need for automated refactoring, in particular for packaging up and breaking down tactic proofs. We also provide some detail about a tactic recording mechanism we have implemented for the HOL

Light system, that acts as solid basis for implementing tool support for proof refactoring, proof graph visualisation and proof querying.

2 Motivation for Tactic Proof Refactoring

We first briefly explain how tactic proofs are often done in practice. When first written, a proof in the subgoal package is usually written as a long series of tactic steps. This first version of a proof is usually not beautiful, but does the job of proving the theorem. What often happens next is that proof is cleaned up, or *refactored*, to become more succint. This refactoring will often involve packaging-up the tactic steps into a single, compound tactic that proves the goal in one step. If done well, the resulting compound tactic is usually neater and more concise because it can factor out repeated use of tactics, for example when the same tactic is applied to each subgoal of a given goal. These packaged-up proofs feature heavily in the source code building up the standard theories of the HOL4 and HOL Light systems, and were a prerequisite for submitting work to the Flyspeck Project for a number of years.

In light of this, automatic refactoring is potentially useful for three reasons. The first is that the process of packaging up a non-trivial proof can be long and tedious, and for proofs that run into dozens of lines it can be easy to miss opportunities to make the proof more concise. Doing this automatically could both save effort and result in more concise proofs.

The second reason is that, given that most of the best examples of tactic proofs are packaged-up, novice users have to laboriously unpick them if they want to step through these masterpieces and learn how the experts prove their theorems. Unpicking a packaged-up proof is even more tedious than packaging it up in the first place, because the user does not know which tactics apply to more than one subgoal and thus need to occur more than once in the unpackaged proof. Automation would improve access to the wealth of experience that is held in source code building up systems' standard theories.

The third reason is that proofs need to be maintained over time, due to changes in the theory context in which the theorems are proved. If the proofs are packaged up, then they will need to be unpackaged, debugged and then repackaged, which again would be considerably easier with automated support.

3 Example Tactic Proof Flattening

In this section we show how automated flattening of a packaged-up tactic proof can reveal the structure of the proof and enable the proof steps to be replayed interactively.

The following is a proof taken from the implementation of HOL Light. It is not a particularly untypical example of such.

```
let REAL_LT_INV = prove
    (`!x. &0 < x ==> &0 < inv(x)`,
     GEN_TAC THEN
     REPEAT_TCL DISJ_CASES_THEN
         ASSUME_TAC (SPEC `inv(x)` REAL_LT_NEGTOTAL) THEN
     ASM_REWRITE_TAC[] THENL
```

```
 [RULE_ASSUM_TAC(REWRITE_RULE[REAL_INV_EQ_0]) THEN
  ASM_REWRITE_TAC[];
  DISCH_TAC THEN
  SUBGOAL_THEN `&0 < --(inv x) * x` MP_TAC THENL
   [MATCH_MP_TAC REAL_LT_MUL THEN ASM_REWRITE_TAC[];
    REWRITE_TAC[REAL_MUL_LNEG]]] THEN
SUBGOAL_THEN `inv(x) * x = &1` SUBST1_TAC THENL
 [MATCH_MP_TAC REAL_MUL_LINV THEN
  UNDISCH_TAC `&0 < x` THEN REAL_ARITH_TAC;
  REWRITE_TAC
     [REAL_LT_RNEG; REAL_ADD_LID; REAL_OF_NUM_LT; ARITH]]);;
```

Unpicking this proof manually is an arduous task. Some tactics get applied to more than one subgoal, but it is not clear which. Furthermore, use of THENL reveals that the proof branches at various points, but it is far from clear which branches are finished within the right hand side of the THENLs and which branches carry on further into the proof script.

The proof actually flattens out into the following series of tactics:

```
e (GEN_TAC);;
e (REPEAT_TCL DISJ_CASES_THEN
      ASSUME_TAC (SPEC `inv x` REAL_LT_NEGTOTAL));;
(* *** Subgoal 1 *** *)
e (ASM_REWRITE_TAC []);;
e (RULE_ASSUM_TAC (REWRITE_RULE [REAL_INV_EQ_0]));;
e (ASM_REWRITE_TAC []);;
(* *** Subgoal 2 *** *)
e (ASM_REWRITE_TAC []);;
(* *** Subgoal 3 *** *)
e (ASM_REWRITE_TAC []);;
e (DISCH_TAC);;
e (SUBGOAL_THEN `&0 < --inv x * x` MP_TAC);;
(* *** Subgoal 3.1 *** *)
e (MATCH_MP_TAC REAL_LT_MUL);;
e (ASM_REWRITE_TAC []);;
(* *** Subgoal 3.2 *** *)
e (REWRITE_TAC [REAL_MUL_LNEG]);;
e (SUBGOAL_THEN `inv x * x = &1` SUBST1_TAC);;
(* *** Subgoal 3.2.1 *** *)
e (MATCH_MP_TAC REAL_MUL_LINV);;
e (UNDISCH_TAC `&0 < x`);;
e (CONV_TAC REAL_ARITH);;
(* *** Subgoal 3.2.2 *** *)
e (REWRITE_TAC
      [REAL_LT_RNEG;REAL_ADD_LID;REAL_OF_NUM_LT;ARITH]);;
```

The flattened proof shows that the application of REPEAT_TCL resulted in three subgoals, with the first subgoal finishing in the first branch of the first THENL, the second subgoal finished off by ASM_REWRITE_TAC prior to the first THENL, and the third subgoal continuing beyond second branch of the first THENL to split and continue to the end of the packaged proof.

4 Requirements for Capturing Tactic Proofs

Underlying a tactic proof refactoring facility must be some system for capturing tactic proofs in a suitable form. This is also true of proof visualisation and proof querying tools. In this section we discuss the requirements for a mechanism that supports all three activities.

First note that it is not particularly important that the original proof can be recreated in all its detail, including those parts that are redundant and don't get executed. For our purposes, we are more interested in what does get executed, which helps proof refactoring, and means that proof visualisation can show what has happened and proof querying can portray what parts of the proof have been used. Thus capturing the proof by a static syntactic transformation of the original proof script will not suit our purposes. Rather, the proof needs to somehow be dynamically recorded, as it is executed, to capture what is actually used. We call this *tactic proof recording*.

Also note that the subgoal package, of course, already dynamically captures tactic proofs, simply as a list of subgoals (or actually a stack of such lists, so that interactive steps can be undone if required). However, this form is not suitable for our purposes because it does not explicitly capture the structure of the proof tree, and neither does it carry the various crucial pieces of information, such as the tactics used in the proof, that we require.

To be most suitable for our purposes of proof refactoring, visualisation and querying, our tactic recording mechanism has seven main requirements:

- To fully capture all the information needed to recreate a proof;
- To capture the parts of the proof that actually get used;
- To capture the information in a form that suitably reflects the full structure of the original proof, including the structure of the goal tree and hierarchy corresponding to the explicit use of tacticals in the proof script;
- To capture information at a level that is meaningful to the user, i.e. with atoms corresponding to the ML binding names for the tactics and other objects mentioned in the proof script;
- To be capable of capturing both complete and incomplete proofs;
- To work both for interactive proofs, possibly spread over several ML commands and involving meta operations for undoing steps or switching between goals, and for non-interactive packaged-up proofs.
- To work for existing proofs, without requiring modification to the original proof script.

5 The Tactic Proof Recording Mechanism

Our recording mechanism is designed to meet the above requirements. It maintains a proof tree in program state, in parallel with the subgoal package's normal state. The proof tree has nodes corresponding to goals in the proof, and branches corresponding to goals' subgoals, reflecting the structure of the original proof. Each node carries information about its goal, including a statement of the goal,

a unique goal identity number and a description of the tactic that got applied to the goal. Active subgoals are labelled as such in place of a tactic description, thus enabling incomplete proofs to be represented. The tree gets added to as interactive tactic steps are executed, and deleted from if steps are undone.

The crucial means by which goals in the subgoal package state are linked to parts of the stored goal tree is based on the goal identity numbers. The datatype for goals is extended to carry such an identity number. These identity-carrying goals are called *xgoals*.

```
type goalid = int;;
type xgoal = goal * goalid;;
```

Tactics are adjusted, or *promoted*, to work with xgoals, so that they take as input an xgoal and return as part of their output a list of uniquely numbered xgoals. When a promoted tactic is applied to an xgoal, its result is incorporated into the proof tree by locating the tree node with the same identity as the tactic's xgoal input. For efficient node location, a separate lookup table is maintained in program state for returning a pointer to the proof tree node corresponding to a given goal identity. The datatype for tactics is a trivial variant of HOL Light's original, with xgoals instead of goals.

```
type xgoalstate = (term list * instantiation) * xgoal list * justification ;;
type xtactic = xgoal -> xgoalstate;;
```

A typical theorem prover has over 100 commonly used tactics. Rather than laboriously implementing promoted forms for each of these, it is preferable to write a generic wrapper function for promoting a supplied tactic. Promoted tactic ML objects overwrite their unpromoted versions, to enable existing proof scripts to be replayed without adjustment.

```
let tactic_wrap name (tac:tactic) : xtactic =
  fun (xg:xgoal) ->
  let (g,id) = dest_xgoal xg in
  let (meta,gs,just) = tac g in
  let obj = Mname name in
  let xgs = extend_gtree id (Gatom obj) gs in
  (meta,xgs,just);;

let REFL_TAC = tactic_wrap "REFL_TAC" REFL_TAC;;
let STRIP_TAC = tactic_wrap "STRIP_TAC" STRIP_TAC;;
let DISCH_TAC = tactic_wrap "DISCH_TAC" DISCH_TAC;;
```

The generic tactic_wrap function promotes tactics of ML datatype tactic. The name argument carries the ML binding name of the tactic that is being promoted. Local value obj is of ML datatype mlobject, for representing the ML expression syntax of the tactic as it occurs in the proof script (see below). In this case, the expression syntax is simply an ML binding name.

It is necessary to write wrapper functions for the datatype of each tactic and inference rule datatype that can occur in a proof script. As the datatypes become more complex, so does the implementation of their corresponding wrapper functions. Slightly more complex than tactic_wrap is term_tactic_wrap, a function for promoting tactics that take a term argument. Its implementation is

similar to tactic_wrap, except that here obj has the expression syntax of an ML name binding applied to a HOL term.

```
let term_tactic_wrap name (tac:term->tactic) (tm:term) : xtactic =
  fun (xg:xgoal) ->
  let (g,id) = dest_xgoal xg in
  let (meta,gs,just) = tac g in
  let obj = Mapp (Mname name, [Mterm tm]) in
  let xgs = extend_gtree id (Gatom obj) gs in
  (meta,xgs,just);;

let UNDISCH_TAC = term_tactic_wrap "UNDISCH_TAC" UNDISCH_TAC;;
let EXISTS_TAC = term_tactic_wrap "EXISTS_TAC" EXISTS_TAC;;
```

The recursive ML datatype mlobject is capable of representing all the ML expression syntax that commonly occurs in tactic proofs, including ML binding names, strings, lists, function applications, HOL terms, etc. A full explanation of this datatype and the implementation of more difficult promotion functions is left for another paper.

```
type mlobject =
  Mname of string
| Mstring of string
| Mlist of mlobject list
| Mapp of (mlobject * mlobject list)
| Mterm of term
| ... ;;
```

6 Conclusion

In this article we have argued the need for a tactic proof refactoring capability, and provided some insight into how tactic proofs can be recorded in HOL Light to support this. The techniques are equally applicable to other subgoal packages implemented in other theorem provers. The same recording mechanism holds information in a form that is ideal for dumping a proof tree graph for proof visualisation, and for proof querying. We intend to implement these facilities as part of the Proof General system [3] for HOL Light.

7 References

[1] L. C. Paulson. Logic and Computation: Interactive proof with Cambridge LCF. (Cambridge University Press, 1987).
[2] T. C. Hales. Introduction to the Flyspeck project. In T. Coquand, H. Lombardi, and M.F. Roy, editors, Mathematics, Algorithms, Proofs, number 05021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs und Forschungszentrum fur Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2006/432.
[3] David Aspinall. Proof General: A Generic Tool for Proof Development. Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS 2000, LNCS 1785.